Estruturas de Dados

Tabelas *Hash*

Luiz Henrique de Campos Merschmann
Departamento de Computação Aplicada
Universidade Federal de Lavras

luiz.hcm@ufla.br

Na Aula de Hoje

Estrutura de Dados: Mapa

Nas aulas anteriores nós conhecemos diferentes estruturadas de dados, ou coleções:

- Lista, Fila e Pilha.

Essas coleções têm uma coisa em comum:

- Em todas elas, cada entrada (ou elemento), é um único valor ou objeto.

Existe também uma estrutura de dados chamada **mapa** que funciona de forma diferente:

- **Cada entrada** dessa estrutura **é um par de objetos chave/valor**.
- Enquanto nas listas um elemento é acessado por posição, em um **mapa** um **valor é acessado a partir** da sua **chave**.

O nome mapa, vem do fato que cada chave *mapeia* um valor.

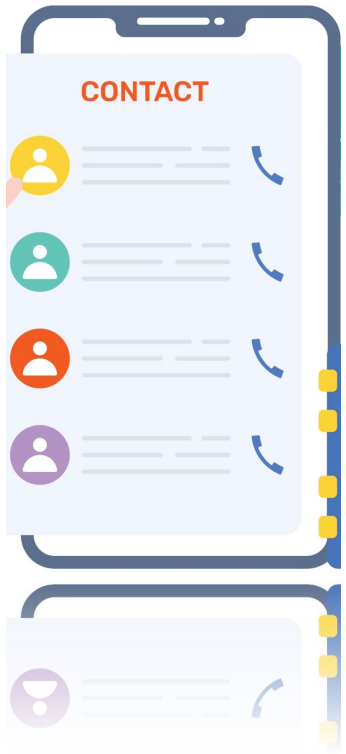
Estrutura de Dados: Mapa

Você consegue pensar em um **exemplo** de mapa, ou seja, coleção de dados organizados na forma de **pares chave/valor**?

A **lista de contatos** do celular é um exemplo de mapa.

- Cada entrada da lista tem um **nome** e um **número de telefone** *(vamos ignorar as demais informações aqui)*.
- Nós **buscamos um valor** (o número do telefone), **a partir de uma chave** (o nome).

Estrutura de Dados Mapa - Exemplo



Nos contatos de um celular, podemos ter uma coleção como a seguinte:

Nome	Número
Tião	9999 1111
Maria	8888 1234
Joaquim	9876 5432

Neste exemplo:

- A primeira entrada tem **a chave** “Tião” **e o valor** “9999 1111”.
- Eu poderia buscar um número a partir da chave “Maria”.
  - E a resposta a esta busca seria “8888 1234”.

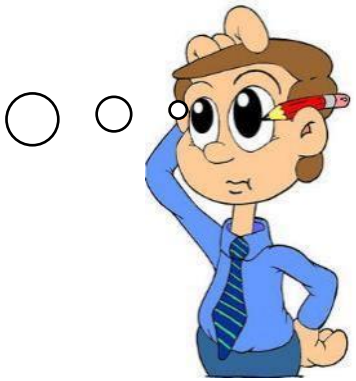
Estrutura de Dados Mapa - Exemplos

Outros exemplos de coleções na forma de mapa são:

- Um **dicionário**:
  - Buscamos um significado (*valor*) a partir de uma palavra (*chave*).
- **Siglas e nomes de estados**:
  - Podemos busca o nome de um estado (*valor*) a partir de sua sigla (*chave*).
- Uma **coleção de alunos**:
  - Podemos buscar um objeto aluno (*valor*) a partir da sua matrícula (*chave*).

Mapa em Java

Mas como podemos usar uma estrutura de dados **Mapa** em um programa **em Java**?



A forma mais simples de usar essa estrutura é através da classe **`HashMap`**.

```
HashMap<String,String> contatos = new HashMap<>();
```

No caso de um mapa, como cada entrada é um par chave/valor, precisamos definir tanto o **tipo da chave** (nesse exemplo é um nome então é String) quanto o **tipo do valor** (o valor é um número de telefone que vamos armazenar como String também).

Nós usamos objetos da classe **HashMap**.

Mapa em Java

Os principais **métodos** que usamos em um objeto HashMap são:

Operação	Método
Adicionar/atualizar entrada (chave c e valor v)	`put(c,v)`
Buscar um valor pela chave	`get(c)`
Remover uma entrada	`remove(c)`
Consultar o número de entradas	`size()`
Consultar se a coleção está vazia	`isEmpty()`
Remover todas as entradas	`clear()`

Note que não podem existir duas entradas com a mesma chave.

- Mas nada impede que duas chaves mapeiem o mesmo objeto.

HashMap em Java: Exemplo

```java
import java.util.HashMap;

public class Programa
{
    public static void main(String[] args)
    {
        HashMap<String,String> contatos = new HashMap<>();

        contatos.put("Tiao", "9999 1111");
        contatos.put("Maria", "8888 1234");
        contatos.put("Joaquim", "9876 5432");

        String telefone = contatos.get("Maria");

        System.out.println("O telefone da Maria é: " + telefone);
    }
}
```

Precisamos importar a classe **HashMap**.

Declaramos e criamos o HashMap conforme já foi mostrado.

Para adicionar entradas no HashMap, usamos o método **put** e informamos a chave e o valor.

Usamos o método **get** para obter um valor a partir de sua chave.

```
O telefone da Maria é: 8888 1234
```

HashMap em Java: Exemplo 2

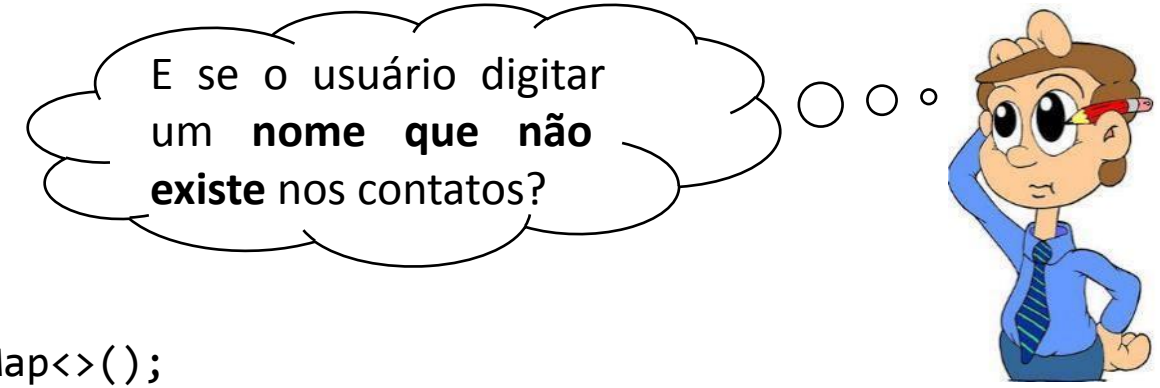
```java
import java.util.HashMap;
import java.util.Scanner;

public class Programa {
    public static void main(String[] args) {

        HashMap<String,String> contatos = new HashMap<>();

        contatos.put("Tiao", "9999 1111");
        contatos.put("Maria", "8888 1234");
        contatos.put("Joaquim", "9876 5432");

        Scanner entrada = new Scanner(System.in);
        System.out.print("Deseja saber o número de quem? ");
        String nome = entrada.nextLine();

        String telefone = contatos.get(nome);
        System.out.println("O telefone de " + nome + " é: " + telefone);
    }
}
```

Fica mais interessante se deixarmos o **usuário escolher** de qual contato ele quer o número de telefone.

```
Deseja saber o número de quem? Joaquim
O telefone de Joaquim é: 9876 5432
```

HashMap em Java: Exemplo 2

```java
import java.util.HashMap;
import java.util.Scanner;

public class Programa {
    public static void main(String[] args) {

        HashMap<String,String> contatos = new HashMap<>();

        contatos.put("Tiao", "9999 1111");
        contatos.put("Maria", "8888 1234");
        contatos.put("Joaquim", "9876 5432");

        Scanner entrada = new Scanner(System.in);
        System.out.print("Deseja saber o número de quem? ");
        String nome = entrada.nextLine();

        String telefone = contatos.get(nome);
        System.out.println("O telefone de " + nome + " é: " + telefone);
    }
}
```




```
Deseja saber o número de quem? Pedro
O telefone de Pedro é: null
```

Veja que o valor retornado é **`null`** E, portanto, se tentarmos chamar um método do objeto retornado, ocorrerá um erro de execução (`java.util.NullPointerException`)

HashMap em Java: Exemplo 3

```java
import java.util.HashMap;
import java.util.Scanner;

public class Programa {
    public static void main(String[] args) {

        HashMap<String,String> contatos = new HashMap<>();

        contatos.put("Tiao", "9999 1111");
        contatos.put("Maria", "8888 1234");
        contatos.put("Joaquim", "9876 5432");
        contatos.put("Maria", "3821 2000");

        Scanner entrada = new Scanner(System.in);
        System.out.print("Deseja saber o número de quem? ");
        String nome = entrada.nextLine();

        String telefone = contatos.get(nome);
        System.out.println("O telefone de " + nome + " é: " + telefone);
    }
}
```

E se tentarmos inserir uma **chave duplicada**?

```
Deseja saber o número de quem? Maria
O telefone de Maria é: 3821 2000
```

Veja que, ao chamar o método **put**, se a chave já existir, a entrada existente é atualizada.

- Ou seja, ele não cria duas entradas com chave duplicada.

HashMap em Java: Exemplo 4

```java
import java.util.HashMap;
import java.util.Scanner;

public class Programa {
    public static void main(String[] args) {

        HashMap<String,String> contatos = new HashMap<>();

        contatos.put("Tiao", "9999 1111");
        contatos.put("Maria", "8888 1234");
        contatos.put("Joaquim", "9876 5432");

        System.out.println("Número de contatos é: " + contatos.size());

        System.out.println("Removendo contato...");
        contatos.remove("Maria");

        System.out.println("Número de contatos é: " + contatos.size());
    }
}
```

Usamos o método **remove** para remover uma entrada a partir de sua chave.

A **remoção** de entradas do HashMap é feita **a partir da chave**.

```
Número de contatos é: 3
Removendo contato...
Número de contatos é: 2
```

HashMap em loops

Não é comum que em um programa precisemos acessar todos os valores de um HashMap.

Mas, caso seja necessário, podemos usar o método **keySet** que retorna o conjunto de chaves do HashMap.

```
HashMap<String,String> contatos = new HashMap<>();

contatos.put("Tiao", "9999 1111");
contatos.put("Maria", "8888 1234");
contatos.put("Joaquim", "9876 5432");

for (String nome : contatos.keySet()) {
    System.out.println("O telefone de " + nome + " é " + contatos.get(nome));
}
```

Esse comando pode ser lido como: "para cada nome presente no conjunto de chaves de contatos, faça".

```
O telefone de Tiao é 9999 1111
O telefone de Joaquim é 9876 5432
O telefone de Maria é 8888 1234
```

A classe HashMap não garante que os elementos estarão na mesma ordem que foram inseridos. Portanto, não podemos percorrer os elementos de um HashMap confiando nisso.

Vantagens do HashMap

Vocês devem se lembrar que na aula sobre `ArrayList`, vimos uma classe `Disciplina` que tinha uma lista de alunos. E como seria se a classe usasse um `HashMap` de pares matrícula/alunos ao invés de um `ArrayList`?

```
private ArrayList<Aluno> alunos;
```

⇒

```
private HashMap<Integer, Aluno> alunos;
```

Podemos usar um **HashMap** no qual a chave é a matrícula e o valor é o aluno.

```
public boolean matricularAluno(Aluno aluno) {
    if (!alunoEstaMatriculado(aluno.getMatricula())) {
        alunos.add(aluno);
        return true;
    }
    return false;
}
```

⇒

```
public boolean matricularAluno(Aluno aluno) {
    if (!alunoEstaMatriculado(aluno.getMatricula())) {
        alunos.put(aluno.getMatricula(), aluno);
        return true;
    }
    return false;
}
```

Usamos o método **put** informando a matrícula do aluno como chave.

```
public Aluno buscarAluno(int matricula) {
    for (Aluno aluno : alunos) {
        if (aluno.getMatricula() == matricula) {
            return aluno;
        }
    }
    return null;
}
```

⇒

```
public Aluno buscarAluno(int matricula) {
    return alunos.get(matricula);
}
```

Com o HashMap não é necessário fazer um loop para encontrar o aluno, basta usar o método **get.**

Vantagens do HashMap

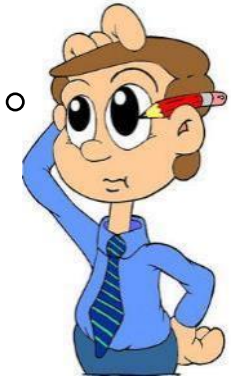
O código ficou mais simples, concorda? Mas a **vantagem** de usar o `HashMap` não é só que o código é mais fácil de implementar:

- Mas especialmente porque essa **busca** é **mais eficiente** (mais rápida).

Veja que no método `buscarAluno`, se usarmos um `ArrayList`, precisamos percorrer a lista de alunos até encontrar aquele que tem a matrícula procurado.

- No pior caso (quando o aluno é o último da lista ou se a matrícula não existir na lista) teremos que percorrer todos os elementos.
  - Imagina se fosse uma lista de alunos da universidade, com 10 mil alunos!

Mas como o `HashMap` consegue fazer buscas mais rapidamente?



Tabelas Hash

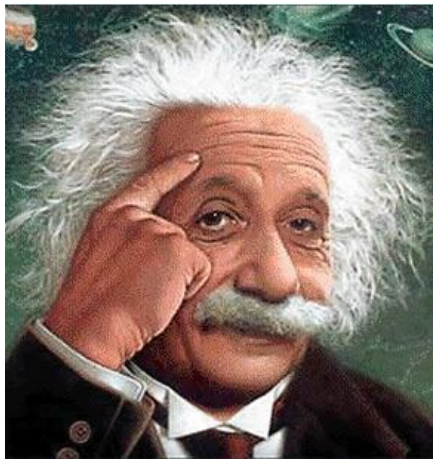
É **essencial** que em uma estrutura de dados **mapa**, os dados sejam organizados de forma que a **busca** de um valor a partir de uma chave seja **fácil e rápida**.

Em exemplos fora da programação, os contatos no celular ou as palavras no dicionário, por exemplo, são organizados em ordem alfabética, o que facilita a nossa busca.

Em programação, uma classe que implementa uma estrutura de dados mapa precisa ter uma forma **eficiente** de **buscar um valor** a partir de uma chave.

- E é para isso que servem as **Tabelas Hash**.
- A classe **`HashMap`** é eficiente, e até tem esse nome, justamente por ser uma **implementação de** mapa feita através de **uma Tabela Hash**.

Tabelas Hash



No nosso exemplo de uma lista de alunos usando um ArrayList, quanto mais alunos tivermos na lista, mais demorada pode ser a busca.

Já as **Tabelas Hash** permitem a realização da **busca em um tempo constante**, **independente do número de elementos** presentes na tabela.

Para se conseguir essa velocidade de busca, uma Tabela Hash usa uma função de espalhamento (***hash function***) para **transformar uma chave C em uma posição na tabela**. Isso significa que:

- Internamente os valores são guardados em um vetor, cujo acesso por posição é muito rápido.
- E a função de *hash* é responsável por calcular qual é a posição de um valor, a partir da sua chave.
- Mas para evitar colisão, ou seja, evitar que duas chaves sejam mapeadas na mesma posição, este vetor costuma ter mais posições do que a quantidade de entradas da tabela.

Obs: a escolha e a implementação de funções de hash eficientes, ou seja, que não usem muita memória adicional e evitem colisões é algo complexo. Nesta disciplina não estudaremos as funções em si. Basta sabermos que a classe HashMap possui uma implementação eficiente e entender suas vantagens.

Exemplo de Disciplina e Alunos

Agora que entendemos porque usar um **HashMap** para a coleção de alunos na classe **Disciplina** é melhor que usar um **ArrayList**, vamos ver o código completo da classe.

Exemplo de Disciplina e Alunos

A classe **Aluno não precisa ser modificada**, veremos as **alterações na classe `Disciplina`**.

```
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;

public class Disciplina {
     private String sigla;
     private String nome;
     private HashMap<Integer, Aluno> alunos;

     public Disciplina(String umaSigla, String umNome) {
          sigla = umaSigla;
          nome = umNome;
          alunos = new HashMap<>();
     }

     public List<Aluno> getAlunosMatriculados() {
          return new ArrayList<Aluno>(alunos.values());
     }

     public boolean alunoEstaMatriculado(int matricula) {
          return alunos.get(matricula) != null;
     }
}
```

As classe `List` e `ArrayList` são necessárias apenas para o método `getAlunosMatriculados`.

O método **`values`** retorna a coleção de valores. Nesse caso, a coleção de alunos.

Para verificar se um aluno está matriculado basta buscar o valor pela chave e verificar se ele é diferente de `null`.

Exemplo de Disciplina e Alunos

```
import java.util.ArrayList; java.util.HashMap;
import java.util.List;
import java.util.Collections; java.util.ArrayList;

public class Disciplina {

     ...

     public boolean matricularAluno(Aluno aluno) {
          // Tratamento para garantir que não pode
          // existir matrícula duplicada na disciplina
          if (!alunoEstaMatriculado(aluno.getMatricula())) {
               alunos.add(aluno); alunos.put(aluno.getMatricula(), aluno);
               return true;
          }
          return false;
     }

     public Aluno buscarAluno(int matricula) {
          for (Aluno a : alunos) {                    return alunos.get(matricula);
               if (a.getMatricula() == matricula) {
                    return a;
               }
          }
          return null;
     }
}
```

Exemplo de Disciplina e Alunos

Esse exemplo é interessante também para reforçar o conceito de **interface de comunicação**.

Repare que nós fizemos **alterações** na estrutura **interna** da classe **`Disciplina`**, mas nós **não alteramos as assinaturas** de seus **métodos**

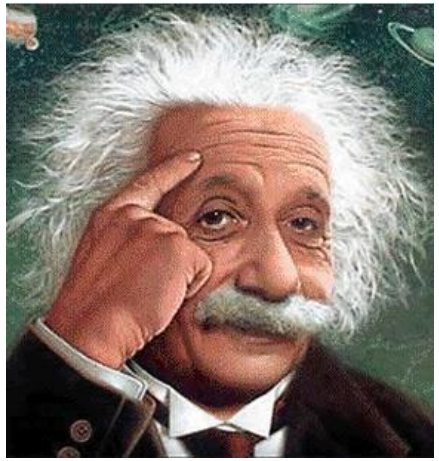
- Ou seja, nós mantivemos a **mesma interface de comunicação**.

Dessa forma, a **classe `Principal`** de nosso programa, que usa a classe `Disciplina`, **não precisa ser alterada.**

- Ela vai continuar funcionando normalmente, sem precisar saber que a classe `Disciplina` foi modificada.

Outras Estruturas de Dados e Coleções

Tabelas Hash



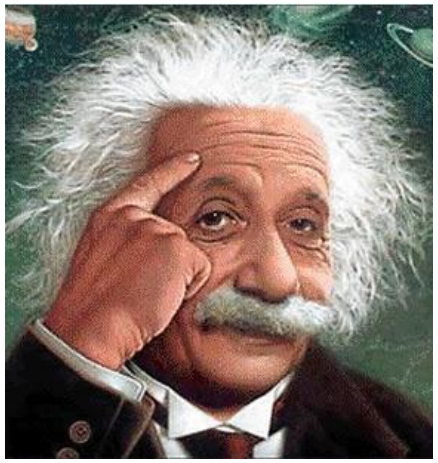
Existem diversas outras **estruturas de dados** que não fazem parte do escopo desta disciplina.

Mas é interessante saber que elas existem e, quando precisar implementar um programa que precisa tratar muitos dados, procurar estudar qual é a melhor para a sua necessidade.

Apenas alguns exemplos de outras coleções disponíveis no Java são:

- **`Set`**: conjunto de elementos que não permite elementos duplicados.
- **`PriorityQueue`**: implementa uma fila na qual os elementos são ordenados por alguma prioridade, ao invés de ser por ordem de chegada.
- **`SortedMap`** e **`SortedSet`**: versões de HashMap e Set que mantêm os valores ordenados.

Tabelas Hash



Além disso, a classe **`Collections`** possui diversos métodos úteis de coleções, especialmente quando usamos uma lista.

Além do método **`unmodifiableList`** que usamos na classe `Disciplina`, alguns outros exemplos são:

- **`min`**: busca o menor valor da coleção.
- **`max`**: busca o maior valor da coleção.
- **`shuffle`**: embaralha aleatoriamente os elementos da coleção.
- **`reverse`**: inverte a ordem dos elementos da coleção.

Perguntas?